Proving Correctness of a Controller Algorithm for the RAID Level 5 System

Mandana Vaziri* Nancy Lynch[a] Jeannette M. Wing

March 1998

CMU-CS-98-117

[a]MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This paper will appear in the *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1998.

This research is sponsored in part by the Defense Advanced Research Projects Agency and the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, F33615-93-1-1330, and Rome Laboratory, Air Force Materiel Command, USAF, F306602-97-2-0031, and in part by the National Science Foundation under Grant No. CCR-9523972. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency Rome Laboratory or the U.S. Government.

Keywords: formal methods, RAID systems, formal specification, verification, correctness proofs, I/O automata

Abstract

Most RAID controllers implemented in industry are complicated and difficult to reason about. This complexity has led to software and hardware systems that are difficult to debug and hard to modify. To overcome this problem Courtright and Gibson have developed a rapid prototyping framework for RAID architectures which relies on a generic controller algorithm. The designer of a new architecture needs to specify parts of the generic controller algorithm and must justify the validity of the controller algorithm obtained. However the latter task may be difficult due to the concurrency of operations on the disks. This is the reason why it would be useful to provide designers with an automated verification tool tailored specifically for the RAID prototyping system. As a first step towards building such a tool, our approach consists of studying several controller algorithms manually, to determine the key properties that need to be verified.

This paper presents the modeling and verification of a controller algorithm for the RAID Level 5 System. We model the system using I/O automata, give an external requirements specification, and prove that the model implements its specification. We use a key invariant to find an error in a controller algorithm for the RAID Level 6 System.

Proving Correctness of a Controller Algorithm for the RAID Level 5 System*

Mandana Vaziri†, Nancy Lynch† and Jeannette Wing‡

†Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

‡Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Most RAID controllers implemented in industry are complicated and difficult to reason about. This complexity has led to software and hardware systems that are difficult to debug and hard to modify. To overcome this problem Courtright and Gibson have developed a rapid prototyping framework for RAID architectures which relies on a generic controller algorithm [1]. The designer of a new architecture needs to specify parts of the generic controller algorithm and must justify the validity of the controller algorithm obtained. However the latter task may be difficult due to the concurrency of operations on the disks. This is the reason why it would be useful to provide designers with an automated verification tool tailored specifically for the RAID prototyping system.

As a first step towards building such a tool, our approach consists of studying several controller algorithms manually, to determine the key properties that need to be verified.

This paper presents the modeling and verification of a controller algorithm for the RAID Level 5 System [5]. We model the system using I/O automata [6], give an external requirements specification, and prove that the model implements its specification. We use a key invariant to find an error in a controller algorithm for the RAID Level 6 System [5].

1 Introduction

Most RAID controllers implemented in industry are complicated and difficult to reason about. This complexity has led to software and hardware systems that are difficult to debug and hard to extend. To overcome this problem Courtright and Gibson have developed a rapid prototyping framework for RAID architectures which relies on a generic controller algorithm [1]. The designer of a new architecture needs to specify parts of the generic controller algorithm and must justify the validity of the controller algorithm obtained. However the latter task may be difficult due to the concurrency of operations on the disks. This is the reason why it would be useful to provide designers with an automated verification tool tailored specifically for the RAID prototyping system.

As a first step towards building such a tool, our approach consists of studying several controller algorithms manually, to determine the key properties that need to be verified.

This paper presents the correctness of a controller algorithm [5] for the RAID Level 5 system. We chose this architecture because of its popularity, as well as for its relative simplicity.

Our method consists of modeling the controller formally, giving an external requirements specification and proving that the model satisfies its specification.

Our study results in a key invariant for the controller that can be generalized for other architectures. We use this invariant to find an error in a RAID Level 6 controller [5].

The outline of the paper is as follows. Section 2 gives background on RAID systems. Section 3 presents the RAID Level 5 system informally. Section 4 gives conventions used throughout the paper. Section 5 describes the specification and Section 6 our model of RAID Level 5. Section 7 presents the proof of correctness and Section 8 the extension of our work to the study of RAID Level 6, as well as the error found. Finally, Section 9 is a summary of our conclusions and future work.

2 RAID Systems

RAID or Redundant Arrays of Inexpensive Disks were developed in the 1980's to address the need for

*This work was supported by DARPA contracts F19628-95-C-0118 and F30602-97-2-0031, and AFOSR contract F49620-97-1-0337.

secondary storage systems with higher performance [9].

A RAID system is composed of a disk array, and a disk array controller. The controller's function is to receive an *operation* from the user of the disk array, and to carry it out by performing a set of *low-level operations* (or *low-level op*) on specific disks.

When the number of disks increases in a disk array, the availability of data and the reliability of the disk array may decrease dramatically [4]. We consider independent catastrophic disk failures, i.e., disk failures in which all data stored on the disk becomes inaccessible and the disk cannot be written any further. RAID systems are designed to be fault-tolerant by storing redundant data [3] on extra disks and to tolerate 1 or 2 disk failures. The redundancy can be an identical copy of each disk, also known as disk mirroring, or it can involve having a *parity disk* [9], for n disks containing data. The parity disk contains blocks, called parity blocks, that cover groups of n blocks independently stored. A set of n blocks along with the parity block that covers them is called a *parity group.* The parity block is computed by performing a bit-wise XOR on the blocks it covers. Given any set of n blocks, the $(n+1)$st block can be recovered by performing an XOR on the n blocks.

There are several RAID architectures that are classified into five "levels" [9]. Different RAID architectures can be distinguished based on their type of encoding, mapping and algorithms used to access data. The encoding indicates the type of redundancy information used, and the mapping the placement of data and redundant information on the disk array.

Algorithms used to access data can be classified as normal-mode and failed-mode ones. In normal-mode, the controller knows about failed disks, if any, and operates on the disk array with this knowledge. In failed-mode, a disk failure has occurred in the middle of controller operation. The controller then needs to recover from the error and complete the operation. This process is called *error recovery.*

Most RAID architectures use *forward error recovery.* This scheme consists of transitioning from a state in which an error has occurred in the middle of controller operation directly to completion. This method involves enumerating a large number of erroneous states. Furthermore it results in architecture-specific controller algorithms, making extension to new architectures difficult.

To overcome this problem, Courtright and Gibson propose a form of backward error-recovery method [1], based on retry[1]. When an error is encountered, the state of the system is modified to note which disk has failed, and the operation is retried based on the new state.

In this approach, operations are represented as Directed Acyclic Graphs (DAGs). Each node in a DAG is a low-level op to be performed on a disk or a low-level op that computes data.

Courtright and Gibson's method of error recovery [1] has two requirements. First, each low-level op must be idempotent, which ensures that a low-level op that is executed several times has the same effect as if it is executed only once. Secondly, the execution of DAGs must leave the array in a *consistent* state, that is the redundant information must be correct.

Due to the concurrency of low-level ops and the existence of failures, reasoning about the correctness of an algorithm using Courtright and Gibson's error recovery method may be difficult. This task would be easier if an automated verification tool were provided.

As a first step towards building such a tool, our approach consists of studying several controller algorithms manually. This paper studies the correctness of a controller algorithm [1] for the RAID Level 5 system [5], that uses Courtright and Gibson's error recovery method. We chose this architecture because of its popularity, as well as for its relative simplicity.

We model the algorithm using I/O automata [7], and give an external requirements specification. We then prove that the model implements its specification, in the sense that there exists a simulation relation [7] from the model to its specification.

3 Informal Description of the RAID Level 5 System

RAID Level 5 [5] uses parity and can tolerate one disk failure. In this architecture, data is block-interleaved and parity blocks are distributed among all the disks in the array. A parity block is the bit-wise XOR of all the blocks it covers. We assume that there are $n+1$ disks in the array.

The controller receives read and write operations from the environment sequentially. Note that each operation requires low-level reads and writes to be performed concurrently on the disks. For each operation it figures out where the data to be read/written is located in the array, and what parity groups are concerned. For each parity group, the controller chooses an algorithm for carrying out the operation and starts executing it. If a disk needed in an algorithm fails

[1]Courtright has since moved on to a similar but different error recovery method called *roll-away error recovery* [2]

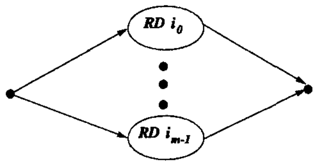

Figure 1: DAG1, Fault Free Read DAG.

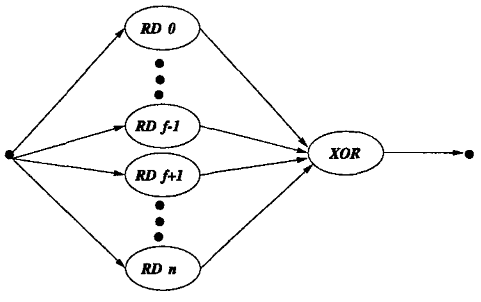

Figure 2: DAG2, Degraded Read DAG.

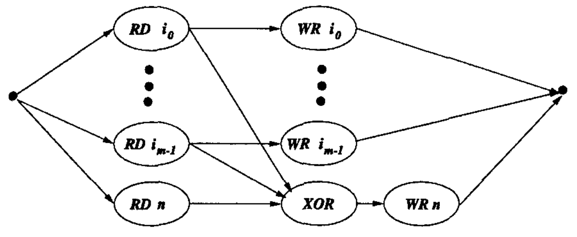

Figure 3: DAG3, Small Write DAG

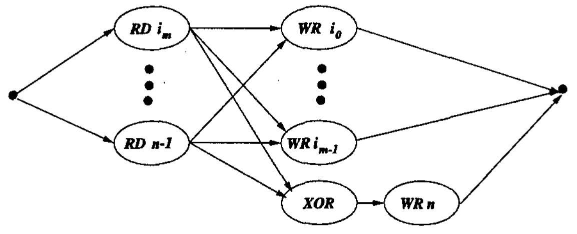

Figure 4: DAG 4, Large Write DAG, no failure

while the algorithm is running, then the controller stops the execution of that algorithm and chooses another one to complete the operation. The controller assumes at most one failure.

Disk array algorithms are represented as Directed Acyclic Graphs (DAGs). Each node in a DAG represents a disk read or write or an XOR. All the low-level ops of a single DAG refer to a unique parity group and represent reads and writes to disk sectors. In a read DAG, we denote the indices of disks to be read by i_0 to i_{m-1} in an arbitrary order, where m is an integer such that $0 \leq m < n$. Similarly, in a write DAG, we denote the indices of disks to be written by i_0 to i_{m-1}. In this case, i_m to i_{n-1} represent indices of disks not to be written. We denote the index of the failed disk by f. In the DAGs, the notation *RD i* means read disk with index i (similarly for *WR i*). DAGs and the criteria for choosing them are described below.

DAGs and DAG selection Figures 1 through 6 present the DAGs. Note that these are informal and do not contain information about how to compute parity blocks. We do not give formal semantics for these DAGs. An arrow from node A to node B indicates that node A must be performed before node B. Low-level ops are atomic. Moreover, if a low-level op fails because of a disk failure, then the DAG stops executing and the controller chooses another DAG to complete the operation.

Fault Free Read (DAG1, Figure 1) The Fault-Free Read DAG is used when there is no failure among the disks to be read. It consists of reading disks containing the data to be read directly.

Degraded Read (DAG2, Figure 2) The Degraded Read DAG is used when one of the disks to be read has failed. It consists of reading the entire array, except the failed disk, and reconstructing the missing data, by taking the bit-wise XOR of the data read.

Small Write (DAG3, Figure 3) The Small Write DAG is used in the absence of failures, when less than half of the array is to be written. In the presence of a failure in a disk that is not to be written, the Small Write is also used regardless of the number of disks to be written. It consists of reading the old data on the disks to be written and the parity, computing the new parity and writing the new parity and the new data. The new parity is the bit-wise XOR of the old data, the new data and the old parity.

Large Write, absence of failure (DAG4, Figure 4) DAG4 is chosen when there are no failures and more than half of the array is to be written. DAG4 consists of reading the data from the disks that are not to be written, computing the parity from the data read and the data to be written and writing the new parity and data. In DAG4, there is an antecedence from each read to each write. Without these antecedences, there exists an execution of the DAG that leaves the disk array in an inconsistent state and no DAG can restore consistency. For example, assume that these antecedences were not there. DAG4 may start by writing a disk, and a disk not to be written fails. In this case, the parity group being modified cannot be restored to a consistent state, because a disk has failed, another one has been modified and the parity contains

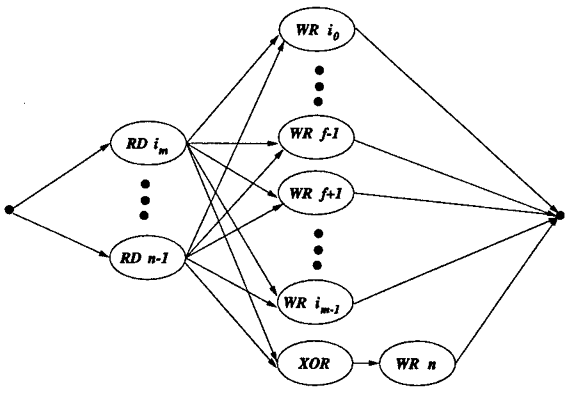

Figure 5: DAG5, Large Write DAG, failure in disk to be written.

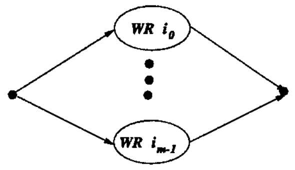

Figure 6: DAG6, Large Write DAG, parity failure.

old information. Thus the antecedences are needed.

Large Write, failure in disk to be written (DAG5, Figure 5) In the presence of a failure in a disk that is to be written, DAG5 is used regardless of the number of disks to be written. DAG5 is identical to DAG4 except that the failed disk is not written.

Large Write, failure in parity disk (DAG6, Figure 6) DAG6 is used when the parity disk has failed. It consists of writing the disks to be written directly.

4 Conventions

In this section, we introduce the notation we use throughout the paper. We use the type $\mathcal{V} = \{0,1\}$ to denote values of bits read or written. The symbol $\perp$ denotes the undefined value, and $\perp^+$ denotes a marked version of the undefined value. The usage of the latter will become apparent when we introduce the models. The type $\mathcal{V}^+$ denotes $\mathcal{V} \cup \{\perp^+\}$. We define the ordering relation $\preceq$ on $\mathcal{V}^+ \cup \{\perp\}$ as follows: $v \preceq v' \Leftrightarrow (v = v') \vee (v \in \{\perp, \perp^+\} \wedge v' \in \{0,1\})$.

We next define types needed for the indices of disks. We use $\mathcal{I}$ to denote the set $\{0, ..., n-1\}$ and $\mathcal{I}_n$ the set $\mathcal{I} \cup \{n\}$. $\mathcal{B}$ is the set of all subsets of $\mathcal{I}$, and $\mathcal{B}_n$ the set of all subsets of $\mathcal{I}_n$. We define the following operators, for $B \in \mathcal{B}$, $B^{co} = \mathcal{I} - B$ and $B_n = B \cup \{n\}$. We use the notation B_n^{co} to denote $(B^{co})_n$. We use the shortcut notation B/j to denote $B/\{j\}$ (where / denotes set difference), for $B \in \mathcal{B}_n$ and $j \in \mathcal{I}_n \cup \{none\}$. In this notation, j intuitively represents a disk that has failed. The value *none* represents no failure. So B/j intuitively means all indices in B except the one that has failed, if any.

Finally, $\mathcal{P}$ is the set of all partial functions from $\mathcal{I}$ to $\mathcal{V}^+$. Likewise, $\mathcal{P}_n$ is the set of all partial functions from $\mathcal{I}_n$ to $\mathcal{V}^+$. We use the symbol $\perp$ to represent the undefined value for a partial function. P_0 of type $\mathcal{P}$ is such that $\forall i \in \mathcal{I}$, $P_0[i] = \perp$. Similarly, P_{n0} of type $\mathcal{P}_n$ is such that $\forall i \in \mathcal{I}_n$, $P_{n0}[i] = \perp$.
We also define the following functions.

- For $P \in \mathcal{P}_n$, $\mathit{indices}(P) = \{i \mid P[i] \in \mathcal{V}\}$, and
- $$\bigoplus P = \begin{cases} \bigoplus_{i \in \mathit{indices}(P)} P[i] & \text{if } \mathit{indices}(P) \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

We also introduce the following shortcuts for the code in our models. For $P \in \mathcal{P}_n$, $B \in \mathcal{B}_n$ and $v \in \mathcal{V}^+$, $P(B) := v$ is equivalent to the following piece of code:
for all $i \in B$ **do** $P[i] := v$ **od** .

For $P \in \mathcal{P}_n$, $Q \in \mathcal{P}$, $P := Q$ is equivalent to the following piece of code:
for all $i \in \mathcal{I}$ **do** $P[i] := Q[i]$ **od** ; $P[n] := \perp$

5 Specification

In this section, we describe the specification, *Spec*, for the system using I/O Automata [7]. *Spec* makes the assumption that there are n bits, indexed from 0 to $n-1$ that can be read or written. It captures the property that the value returned on a read from a bit corresponds to the last write to that bit, or an arbitrary value if no write has been performed. It interacts with an environment automaton *Env*, which requests one operation at a time. We use *Spec'* to denote the composition of *Spec* with *Env*. Thus the inputs in *Spec'* happen one at a time, and each input is submitted until after the previous one has been processed.

Figure 7 presents the model for the specification. Underlined variables and statements are not part of the basic model. These variables are history variables and will be introduced below. The model has the external interface shown in Figure 7.

Spec has the following state variables. *Bit* is an array of n bits, indexed from 0 to $n-1$, initially arbitrary. *ReadPairs* is of type $\mathcal{P}$ and is initially P_0. It is used to record what bits need to be read. *WritePairs* is of type $\mathcal{P}$ and is initially P_0. It is used to record what bits and values need to be written. *pc* ranges over $\{idle, read, write\}$ and is initially *idle*.

Spec works as follows. *Read(B)* is an input from the environment. It has the effect of placing a placeholder $\perp^+$ in *ReadPairs* for every index that needs

```
Spec

Signature

Inputs:
    Read(B)              $B \in \mathcal{B}$
    Write(P)             $P \in \mathcal{P}$
Internals:
    read(i)              $i \in \mathcal{I}$
    write(i,v)           $i \in \mathcal{I}$, $v \in \mathcal{V}$

Outputs:
    ReadBack(P)          $P \in \mathcal{P}$
    WriteOK

State

    Bit                  Array of n bits, indexed from 0 to n-1,
                         initially arbitrary.
    ReadPairs            $\mathcal{P}$, initially $P_0$.
    WritePairs           $\mathcal{P}$, initially $P_0$.
    pc                   {idle,read,write}, initially idle.

    $\underline{Indices}$              $\mathcal{B}$, initially empty.
    $\underline{WritePairsPerm}$       $\mathcal{P}$, initially $P_0$.

Transitions
 Read(B)
  Eff:
   $ReadPairs(B) := \perp^+$
   $pc := read$
   $\underline{Indices := B}$

 read(i)
  Pre:
   $pc = read$
   $ReadPairs[i] = \perp^+$
  Eff:
   $ReadPairs[i] := Bit[i]$

 ReadBack(P)
  Pre:
   $pc = read$
   $\forall i \in \mathcal{I}, ReadPairs[i] \neq \perp^+$
   $P = ReadPairs$
  Eff:
   $ReadPairs := P_0$
   $pc := idle$
   $\underline{Indices := \{\}}$

 Write(P)
  Eff:
   $WritePairs := P$
   $pc := write$
   $\underline{Indices := indices(P)}$
   $\underline{WritePairsPerm := P}$

 write(i,v)
  Pre:
   $pc = write$
   $WritePairs[i] = v$,
      where $v \in \mathcal{V}$
  Eff:
   $WritePairs[i] := \perp$
   $Bit[i] := v$

 WriteOK
  Pre:
   $pc = write$
   $WritePairs = P_0$
  Eff:
   $pc := idle$
   $\underline{Indices = \{\}}$
   $\underline{WritePairsPerm := P_0}$
```

Figure 7: I/O Automaton for *Spec.*

to be read. Upon receiving a *Read*(B) input, the automaton performs a series of internal *read*(i) actions to perform the read. Each such action reads *Bit*[i] and records that value in *ReadPairs.* When all bits that had to be read have been read, the automaton performs a *ReadBack*(P) output, where P is identical to *ReadPairs.* This action resets state variables. *Write*(P) is the other input from the environment. It has the effect of setting the variable *WritePairs* to P, thereby recording which bits need to be written with what values. Upon receiving a *Write*(P) input, the automaton performs a series of internal *write*(i,v) actions to perform the writes. Each such action writes value v to *Bit*[i] and sets *WritePairs*[i] to $\perp$. When *WritePairs* has no more values to be written, the automaton performs a *WriteOK* output, which sets *pc* back to *idle.*

We add the following history variables to *Spec.* The modified automaton is shown in Figure 7. The changes are shown by underlining. These variables are useful for the main proof of correctness. The first one is *Indices* of type $\mathcal{B}$, initially empty. Actions *Read*(B) and *Write*(P) set it and *ReadBack*(P) and *WriteOK* reset it. It records what bits need to be read during a read operation and what bits need to be written during a write. The second one is *WritePairsPerm* of type $\mathcal{P}$, initially P_0. It records bits and values to be written during a write operation. It does not change as the write progresses. Actions *Write*(P) and *WriteOK* modify it.

6 Model for the RAID Level 5 System

In this section, we give our model for the RAID Level 5 System. The model makes the following assumptions. First, the controller uses the same set of DAGs on every parity group and two DAGs on different parity groups do not interfere with each other's execution. Therefore it is sufficient to show that the controller's behavior is correct on one parity group. Thus the model assumes a single parity group consisting of $n+1$ disks, indexed from 0 to n, where disk n is the parity disk. Secondly, all parity computations are bit-wise. Therefore the model assumes one bit per disk, and this restriction can be removed trivially. Finally, the model assumes at most one disk failure.

We present the RAID Level 5 model in Figures 8 and 9, and refer to this model as *RAID.* In the figures, underlined variables and statements are not part of the basic model. These variables are history variables and will be introduced below.

The external interface of *RAID* is the same as the one for *Spec. RAID* interacts with the same environment *Env* as *Spec. Env* requests one operation at a time. We use *RAID'* to denote the composition of *Env* and *Spec.*

RAID has the following state variables. *Disk* is an array of *n+1* bits, indexed from *0* to *n.* This variable models a single parity group in a physical RAID system. *Disk*[n] models the parity. The values in *Disk* are initially such that: $\bigoplus_{0 \leq i \leq n} Disk[i] = 0$. *Indices* is of type $\mathcal{B}$, initially empty, and is used to hold the indices of disks to be written or read. *ReadPairs* is of type $\mathcal{P}_n$, initially P_{n0}. It is used to remember values read from *Disk.* It is also used to indicate which indices need to

RAID

Signature

Inputs:

$Read(B)$	$B \in \mathcal{B}$
$Write(P)$	$P \in \mathcal{P}$

Internals:

$read(i)$, $fail(i)$	$i \in \mathcal{I}_n$
$write3(i,v)$, $write45(i,v)$, $write6(i,v)$	$i \in \mathcal{I}_n$, $v \in \mathcal{V}$
$chooseDAG1$, $chooseDAG2$	
$chooseDAG3$, $chooseDAG4$	
$chooseDAG5$, $chooseDAG6$	

Outputs:

$ReadBack(P)$	$P \in \mathcal{P}$
$WriteOK$	

State

$Disk$	Array of $n+1$ bits, indexed from 0 to n, initially such that $\bigoplus_i Disk[i] = 0$.
$Indices$	$\mathcal{B}$, initially empty.
$ReadPairs$	$\mathcal{P}_n$, initially P_{n0}.
$WritePairs$	$\mathcal{P}_n$, initially P_{n0}.
$WritePairsPerm$	$\mathcal{P}$, initially P_0.
DAG	$\{none, chooseR, chooseW, 1, 2, 3, 4, 5, 6\}$, initially $none$
f	$\mathcal{I}_n \cup \{none\}$, initially $none$.
$\underline{VD}$	$\mathcal{V}$, initially 0.
$\underline{Rec}$	Boolean, initially $false$.

Transitions

```
Read(B)
  Eff:
  $Indices := B$
  $ReadPairs(B) := \perp^+$
  $DAG := chooseR$

chooseDAG1
  Pre:
  $DAG = chooseR$
  $f \notin Indices$
  Eff:
  $DAG := 1$

chooseDAG2
  Pre:
  $DAG = chooseR$
  $f \in Indices$
  Eff:
  $DAG := 2$
  $ReadPairs(Indices_n^{co}) := \perp^+$

read(i)
  Pre:
  $DAG \in \{1,2,3,4,5\}$
  $i \neq f$
  $ReadPairs[i] = \perp^+$
  Eff:
  $ReadPairs[i] := Disk[i]$

ReadBack(P)
  Pre:
  $DAG \in \{1,2\}$
  $\forall i \in Indices/f,$
      $P[i] = ReadPairs[i]$
  $\forall i \in Indices^{co},$
      $P[i] = \perp$
  $\forall i \in Indices/f,$
      $ReadPairs[i] \in \mathcal{V}$
  if $DAG = 2$
  then
      $\forall i \in Indices_n^{co}$
          $ReadPairs[i] \in \mathcal{V}$
      $P[f] = \bigoplus ReadPairs$
  fi
  Eff:
  $ReadPairs := P_{n0}$
  $Indices := \{\}$
  $DAG := none$
  $\underline{Rec := false}$
```

Figure 8: I/O Automaton for *RAID* with history variables.

be read. *WritePairs* is of type $\mathcal{P}_n$, initially P_{n0}. It is used to remember values that are to be written to *Disk* and that have not been written yet. *WritePairsPerm* is of type $\mathcal{P}$, initially P_0. It is used to remember the values that are to be written to disks other than the parity, for the duration of a write operation. *DAG* ranges over $\{none, chooseR, chooseW, 1, 2, 3, 4, 5, 6\}$ and indicates which DAG is currently running or whether a DAG is to be chosen. It is set to *none* when the automaton is idle, which is also its initial value. *f* is of type $\mathcal{I}_n \cup \{none\}$ and indicates which disk has failed. If it is equal to *none*, then no disk has failed. A failed disk cannot be read or written any further. This models the catastrophic failure of disks. *f* is initially *none.*

We define the following derived variables for *RAID*, which are used in the statement of properties in subsequent sections: $All = \mathcal{I}_n/f$, and *StartedWriting*, is a boolean equivalent to $\exists i \in Indices$, $WritePairs[i] \neq WritePairsPerm[i]$.

RAID does not represent DAGs explicitly. DAG nodes are represented by the actions *read*(i), *write3*(i,v), *write45*(i,v) and *write6*(i,v). DAG precedences are represented in the preconditions of low-level writes. DAG selection is done using actions *chooseDAG1*, through *chooseDAG6*. Note that XOR nodes in the DAGs are not present in the model as separate actions, the XORs are computed in the preconditions of low-level writes to *Disk*$[n]$.

We now explain how *RAID* works. When *RAID* receives a *Read*(B) input, it records which indices are to be read in the variable *Indices.* It also sets *ReadPairs*$[i]$, for $i \in B$, to $\perp^+$. The symbol $\perp^+$ is used as a placeholder. The values read from the disk array will be placed into this state variable and $\perp^+$ will be replaced with values read. When *RAID* receives a *Write*(P) input, it records which indices are to be written along with corresponding values in variable *WritePairs.*

After receiving an input, *RAID* proceeds to choosing a DAG to execute. The selection criteria appear in the precondition of each *chooseDAG* action, and is the same as what we described in Section 3.

chooseDAG actions may change *ReadPairs* by putting placeholders for indices to be read. They may also change *WritePairs*$[n]$ to $\perp^+$, which signifies that *Disk*$[n]$ needs to be written. When *Disk*$[n]$ is actually written, then *WritePairs*$[n]$ is set back to $\perp$.

After a DAG has been chosen it executed by performing low-level read and write actions. The precondition for a low-level read action includes *ReadPairs*$[i]$ $= \perp^+$, which means that the read needs to be performed and has not been performed yet. The action records the value read in place of the placeholder $\perp^+$.

The preconditions for low-level write actions include $WritePairs[i] \in \mathcal{V}$, if $i \neq n$, and $WritePairs[i] = \perp^+$, if $i = n$. These expressions indicate that the low-

```
Write(P)
  Eff:
  WritePairs := P
  WritePairsPerm := P
  Indices := indices(P)
  DAG := chooseW

chooseDAG3
  Pre:
  DAG = chooseW
  f = none ∧ |Indices| ≤ n/2
  ∨ f ∈ $Indices^{co}$
  Eff:
  DAG := 3
  ReadPairs($Indices_n$) := $\bot^+$
  WritePairs[n] := $\bot^+$

chooseDAG4
  Pre:
  DAG = chooseW
  f = none ∧ n/2 < |Indices|
  Eff:
  DAG := 4
  ReadPairs($Indices^{co}$) := $\bot^+$
  WritePairs[n] := $\bot^+$

chooseDAG5
  Pre:
  DAG = chooseW
  f ∈ Indices
  Eff:
  DAG := 5
  ReadPairs($Indices^{co}$) := $\bot^+$
  WritePairs[n] := $\bot^+$

chooseDAG6
  Pre:
  DAG = chooseW
  f = n
  Eff:
  DAG := 6

write3(i,v)
  Pre:
  DAG = 3
  $i \neq n$
  $i \neq f$
  WritePairs[i] = v, $v \in \mathcal{V}$
  ReadPairs[i] $\neq \bot^+$
  Eff:
  WritePairs[i] := $\bot$
  Disk[i] := v

WriteOK
  Pre:
  DAG ∈ {3,4,5,6} ∧
  WritePairs = $P_{n0}$
  Eff:
  Indices := {}
  ReadPairs := $P_{n0}$
  WritePairsPerm := $P_0$
  DAG := none
  $\underline{Rec := false}$
  $\underline{\textbf{if } f \in Indices \textbf{ then}}$
    $\underline{VD :=}$
      $\underline{WritePairsPerm[f] \textbf{ fi}}$

write3(n,v)
  Pre:
  DAG = 3
  $n \neq f$
  WritePairs[n] = $\bot^+$
  v= $\bigoplus$ReadPairs
    $\oplus \bigoplus$WritePairsPerm
  $\forall i \in Indices_n$
    ReadPairs[i] $\neq \bot^+$
  Eff:
  WritePairs[n] := $\bot$
  Disk[n] := v

write45(i,v)
  Pre:
  DAG ∈ {4,5}
  $i \neq n$
  $i \neq f$
  WritePairs[i] = v, $v \in \mathcal{V}$
  $\forall i \in Indices^{co}$
    ReadPairs[i] $\neq \bot^+$
  Eff:
  WritePairs[i] := $\bot$
  Disk[i] := v

write45(n,v)
  Pre:
  DAG ∈ {4,5}
  $n \neq f$
  WritePairs[n] = $\bot^+$
  v = $\bigoplus$ReadPairs
    $\oplus \bigoplus$WritePairsPerm
  $\forall i \in Indices^{co}$
    ReadPairs[i] $\neq \bot^+$
  Eff:
  WritePairs[n] := $\bot$
  Disk[n] := v

write6(i,v)
  Pre:
  DAG = 6
  $i \neq n$
  WritePairs[i] = v, $v \in \mathcal{V}$
  Eff:
  WritePairs[i] := $\bot$
  Disk[i] := v

fail(i)
  Pre:
  f = none
  Eff:
  f := i
  if ReadPairs[i] = $\bot^+$
    ∨ WritePairs[i] $\neq \bot$
  then
    if DAG ∈ {chooseR,1}
    then
      ReadPairs(Indices) := $\bot^+$
      DAG := chooseR
    fi
    if DAG ∈ {chooseW,3,4}
    then
    ReadPairs := $P_{n0}$
    WritePairs :=
      WritePairsPerm
    DAG := chooseW
    fi
    $\underline{Rec := true}$
  fi
  $\underline{\textbf{if } i \neq n \textbf{ then}}$
    $\underline{VD := Disk[i] \textbf{ fi}}$
```

Figure 9: I/O Automaton for *RAID* with history variables *(Continued).*

level write needs to be performed and has not been performed yet. The action has the effect of setting *WritePairs*[*i*] to $\bot$. The precondition of these actions also encodes the precedences in DAGs.

If *DAG* ∈ {1,2} and all the appropriate low-level reads have been performed, then the controller performs a *ReadBack*(*P*) output. In the case of DAG2, this action computes the value of disk *f* by taking the XOR of every value in *ReadPairs*. This variable has a value in $\mathcal{V}$ for every $i \in \mathcal{I}_n$ except *f*. Therefore the value computed for disk *f* is the XOR of every other disk. The effect of the action is to reset state variables.

If *DAG* ∈ {3,4,5,6} and all the appropriate low-level writes have been performed, then the controller performs a *WriteOK* action. Its effect is to reset state variables.

A *fail*(*i*) action may occur at most once. It sets variable *f* to *i*. The test *ReadPairs*[*i*] = $\bot^+$ ∨ *WritePairs*[*i*] $\neq \bot$ checks whether there is any low-level read or write on *Disk*[*i*] that needs to be executed. In that case, the action stops the execution of the current DAG by changing *DAG* to either *chooseR* or *chooseW*, which causes a new DAG to be chosen using the same rules as before. Otherwise, *DAG* remains unchanged.

Finally, we add the following history variables to *RAID*. The changes are shown in Figures 8 and 9 by underlining. The first one is *VD* of type $\mathcal{V}$, which stands for Virtual Disk and is initially 0. It is used to keep the last value written to a disk, except the parity disk, if it has failed. *VD* is updated as indicated in the figure. The second one is *Rec*, which is a boolean, initially *false*. It is *true* when a DAG has stopped execution because of a failure and a second DAG is running to complete the operation. *Rec* is updated by *fail*(*i*), *ReadBack*(*P*) and *WriteOK*.

7 Proof of Correctness

This section presents the proof of correctness. We show that *RAID′* implements *Spec′*, by showing that there exists a simulation relation [7] from *RAID′* to *Spec′*. Section 7.1 presents the key invariant called *consistency*. Section 7.2 gives the simulation relation to be proved. Section 7.3 gives the step correspondence of the proof. The complete formal proof of the simulation relation is not presented here and appears in [10].

7.1 Consistency

In this section, we present the consistency property. A proof of this Lemma appears in [10]. Informally, a parity group with no failure is consistent, if the XOR of all bits is equal to 0. If there is a failure at a disk other that the parity disk, then the XOR of all bits, except the one that has failed, is equal to the last value written to the failed bit. Thus consistency can be expressed as: If $f \neq$ n, then $\bigoplus_{i \in All} Disk[i] = VD$. Note that if there is no failure then $VD = 0$.

Lemma 7.1 Consistency *In all reachable states of RAID′, if $n \neq f$, then:*

1. *If $DAG \in \{none, chooseR, 1, 2\} \vee DAG = chooseW \wedge f \notin Indices \vee DAG = 4 \wedge \neg StartedWriting$, then $\bigoplus_{i \in All} Disk[i] = VD$.*
2. *If $DAG \in \{3,5\} \vee DAG = 4 \wedge StartedWriting$, then $\bigoplus ReadPairs \oplus \bigoplus_{ReadPairs[i]=\perp^+} Disk[i] \oplus \bigoplus_{i \in Indices^{co}/f} Disk[i] = \begin{cases} VD & \text{if } f \notin Indices \\ 0 & \text{otherwise} \end{cases}$*

Lemma 7.1 consists of two parts. The first part expresses under what conditions the parity group is consistent. These conditions are: when the controller is idle or doing a read operation, when the controller is about to choose a write DAG and the failed disk, if any, is not among the disks to be written, and when DAG4 is executing and it has not started writing. This last condition is true because of the dependencies in DAG4 that force all the writes to occur after all the reads.

When the controller is executing a write DAG and it has started writing, the parity group is no longer consistent. The second part of Lemma 7.1 expresses an invariant relevant to the execution of write DAGs (3,4,5) that is needed to restore the parity group to a consistent state when the write operation is done. Note that the parity group is trivially consistent after the execution of DAG6, since $f = n$ in that case.

The second property expresses the fact that during the execution of a write DAG, the XOR of the disks read, the disks to be read, and the disks not to be written except for the failed one, is equal to *VD* if there is no failure among disks to be written, and 0 otherwise. The second property is a technical invariant needed to prove the first.

7.2 Simulation Relation

In this section, we give a relation between states of *RAID′* and *Spec′*. We need to show that this relation is a simulation relation. Let s and u be states of *RAID′* and *Spec′* respectively, and f the following relation.

$f(s,u) \Leftrightarrow f_1(s,u) \wedge f_2(s,u) \wedge f_3(s,u) \wedge f_4(s,u) \wedge f_5(s,u)$, where $f_1(s,u)$ through $f_5(s,u)$ are defined below.

- $f_1(s,u) \Leftrightarrow \forall\ i$ **s.t.** $0 \leq i <$ n
  if $s.f \neq i$ **then** $u.Bit[i] = s.Disk[i]$
  else $u.Bit[i] = s.VD$
- $f_2(s,u) \Leftrightarrow \forall i \in s.Indices$
  if $s.DAG \in \{chooseR, 1, 2\} \wedge$
  $(s.Rec = false \vee i = s.f)$
  then $s.ReadPairs[i] = u.ReadPairs[i]$
  else $s.ReadPairs[i] \preceq u.ReadPairs[i]$
- $f_3(s,u) \Leftrightarrow \forall i \in s.Indices$
  if $s.DAG \in \{chooseW, 3, 4, 5, 6\} \wedge$
  $(s.Rec = false \vee i = s.f)$
  then $u.WritePairs[i] = s.WritePairs[i]$
  else $u.WritePairs[i] \preceq s.WritePairs[i]$
- $f_4(s,u) \Leftrightarrow$ **if** $s.DAG = none$ **then** $u.\text{pc} = idle$
  elseif $s.DAG \in \{chooseR, 1, 2\}$ **then** $u.\text{pc} = read$
  else $u.\text{pc} = write$
- $f_5(s,u) \Leftrightarrow u.WritePairsPerm = s.WritePairsPerm \wedge u.ready = s.ready$.

f_1 gives the correspondence between *Bit* and *Disk* variables. If disk i has failed, then $u.Bit[i] = s.VD$. f_2 and f_3 give the correspondence for *ReadPairs* and *WritePairs* variables. If the controller is running a DAG right after receiving an input ($s.Rec = false$), then the variables are equal to their counterparts in *Spec*. On the other hand, if a DAG has failed and the controller is running a second DAG to complete the operation then the variables are related to their counterparts with the $\preceq$ relation, defined previously in Section 4. In either case, if a disk i has failed, then these variables evaluated at i are equal to their counterparts. f_4 gives the correspondence between *DAG* and *pc*. f_5 gives some trivial equalities.

Theorem 7.2 *f is a simulation relation from RAID′ to Spec′.*

7.3 Step Correspondence

In order to prove that f is a simulation relation, we need to show that each execution of *RAID′* has a corresponding execution in *Spec′* having the same external actions. For each transition of *RAID′* we need to give the corresponding sequence of steps in *Spec′*. Let s and u be reachable states of *RAID′* and *Spec′*, respectively, such that $f(s,u) = true$, and (s,π,s') is a transition of *RAID′*.

- $\pi \in \{Read(B), Write(P)\}$. Let the corresponding execution fragment be $\underline{u, Read(B), u'}$, and $\underline{u, Write(P), u'}$, respectively.
- $\pi = read(i)$. If $s.DAG \in \{3,4,5\} \vee s.DAG = 2 \wedge i \notin s.Indices$.
  Let the corresponding execution fragment be $\underline{\text{none}}$. In this case, the value read by π is not

directly returned to *Env*. These reads are performed to compute parity or the value of a lost data being read.

If $s.DAG = 1 \lor s.DAG = 2 \land i \in s.Indices$. In this case, the value read by π needs to be returned directly to *Env*. Thus *Spec'* needs to perform a *read*(i). However it could be that *RAID'* is running a second DAG for the current operation and that a *read*(i) has already been performed, in which case *Spec'* has also already performed a *read*(i). In this case, $u.ReadPairs[i] \in \mathcal{V}$, and the corresponding execution fragment is none. Otherwise it is u,*read*(i),u'.

- $\pi \in \{write3(i,v), write45(i,v), write6(i,v)\}$, where $i \neq n$. *Spec'* also needs to perform *write*(i,v). However it could be that *RAID'* is running a second DAG for the current operation and that π has already been performed, in which case *Spec'* has also already performed a *write*(i,v). In this case, $u.WritePairs[i] = \bot$, and the corresponding execution fragment is none. Otherwise, it is u,*write*(i,v),u'.

- π = *ReadBack*(P). If $s.DAG = 1$, then let the corresponding execution fragment be u,*ReadBack*(P),u'.

  If $s.DAG = 2$, then let the corresponding execution fragment be u,*read*($s.f$),u'',*ReadBack*(P),u'. In this case, there is a failure among disks to be read. In *RAID'*, reads of failed disks do not occur. But the value needs to be read in *Spec'*. We use the consistency property presented previously, to argue that *RAID'* returns the right value for the failed disk.

- π = *WriteOK*. If $s.DAG \in \{3,4,6\}$, then let the corresponding execution fragment be u, *WriteOK*,u'.

  If $s.DAG = 5$, then let the corresponding execution fragment be u,*write*($s.f$,v),u'', *WriteOK*,u', where v = *WritePairsPerm*[f]. In this case, there is a failure among disks to be written. In *RAID'*, writes to failed disks do not occur. But the write needs to be performed in *Spec'*.

- $\pi \in \{fail(i),\ chooseDAG1 - chooseDAG6,\ write(n,v)\}$. Let the corresponding execution fragment be none.

Recall from the introduction that, informally, the two conditions for correctness are consistency and idempotency. We indicate in what follows where these

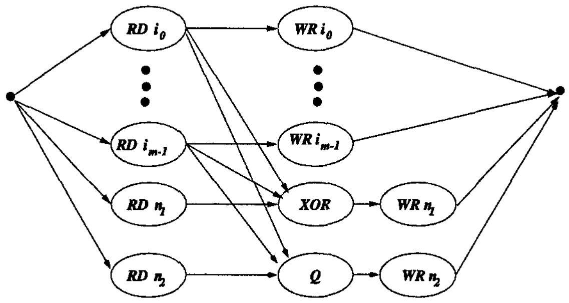

Figure 10: RAID Level 6 - Small Write.

properties are used. Consistency is used to give the step correspondence for *ReadBack*(P), in the case where there is a failure among disks to be read. Idempotency of read (write) low-level ops is used for the step correspondence of *read*(i) (*write*(i,v)), in the case where *RAID'* is running a second DAG to complete an operation and this low-level op has been performed once before.

8 Extension

We now turn our attention to a controller algorithm for the RAID Level 6 architecture [5]. We generalize part of the main invariant, consistency, and use it to find an error in a RAID Level 6 DAG.

8.1 RAID Level 6

RAID Level 6 uses two parity blocks for each group of n blocks stored on separate disks. It can tolerate two disk failures. One parity block (n_1) is computed by taking the XOR of all data blocks. The other parity (n_2) is computed using Reed-Solomon codes.

RAID Level 6 uses Courtright and Gibson's error recovery method analogously to RAID Level 5. It has DAGs that are similar. In particular the Small Write DAG is shown in Figure 10. The symbol Q indicates the computation of n_2.

8.2 Error found

We use a generalized version of part of the consistency Lemma to find an error in the Small Write DAG, without performing the entire proof of correctness for RAID Level 6. Consider the following part of the consistency Lemma (Invariant 1): In all reachable states of *RAID'*, if $n \neq f$ and $DAG = chooseW \land f \notin Indices$, then $\bigoplus_{i \in All} Disk[i] = VD$.

Consider the case in which there is a failure in a disk not to be written. Intuitively, this invariant says that if the controller is about to choose a write DAG, then the value implied by the system for the failed disk ($\bigoplus_{i \in All} Disk[i]$) is equal to the value of the last write to that disk (VD).

Consequently, for this invariant to be true, it must be that if a write DAG fails in such a way that there is a failure among the disks not to be written, then the value implied by the system for the failed disk is equal to the value last written to it. In other words, the failure of a write DAG should not cause the loss of data in disks not to be written.

We use this idea to find an error in the DAG presented above. Consider a scenario in which, *RD* i_0 and *WR* i_0 are performed, then disk indexed i_m (not to be written) fails. This does not cause the DAG to stop. But suppose disk i_1 then failed as well. In this case, the DAG stops and a new DAG needs to be chosen to complete the operation. However the value of disk i_m has been lost, because the array has been partially updated. In addition, when a DAG fails the state is reset and thus it is impossible to recover the value of the failed disk.

9 Summary and Future Work

In this paper, we used I/O Automata to model and verify a controller algorithm for the RAID Level 5 system, which uses Courtright and Gibson's error recovery method. By performing this case study, we formalized a key invariant, consistency, which helped in finding an error in a different more complicated RAID controller algorithm.

This project started out by a preliminary study using the model checker SMV [8]. We modeled the DAGs for RAID Level 5 separately and used the tool to show that the DAGs preserve consistency. However it became clear that our notion of consistency was not accurate and that we needed to formalize this property. This led us to the study of the controller algorithm as a whole.

With the formalization of the consistency invariant we can envisage a tool that takes a model of a controller algorithm based on Courtright and Gibson's error recovery, and checks that consistency is preserved in all reachable states. Such a tool can be built based on a model checker.

The advantage of such a tool is that it would be specifically tailored to Courtright and Gibson's prototyping system. Its users will not need to know about the formalization of the consistency property and will not need to reproduce the hand-proof present in this case study. However hand-proofs are essential at this stage of the design of the tool, because they allow us to determine the exact expression of properties to verify.

Courtright credits our work in his PhD thesis [2] as playing a role in debugging his designs and he encourages continued work in this direction, especially in collaboration with industry partners.

Future work consists of proving correctness of other RAID controllers using Courtright and Gibson's error recovery, as well as considering controller algorithms that use Courtright's latest error recovery method [2]. Finally, we plan to build a special-purpose verification tool.

References

[1] W. V. Courtright II and G. A. Gibson. "Backward error recovery in redundant disk arrays." *Proceedings of the 20th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems (CMG).* December 4-9 1994, pp. 63–74.

[2] William V. Courtright II, "A Transactional Approach to Redundant Disk Array Implementation." Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, Ph.D. thesis, April 1997.

[3] Garth Gibson. "Redundant Disk Arrays: Reliable, Parallel Secondary Storage". PhD thesis, University of California at Berkeley, 1990. Report UCB/CSD 91/613.

[4] G. A. Gibson and D. A. Patterson, "Designing disk arrays for high data reliability", *Journal of Parallel and Distributed Computing.* 17(1-2), 1993, 4-27.

[5] G. Gibson, W. Courtright II, M. Holland, and J. Zelenka, "RAIDframe: Rapid prototyping for disk arrays," Computer Science Technical Report CMU-CS-95-200, Carnegie Mellon University, 1995.

[6] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. *CWI-Quaterly,* 2(3): 219-246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[7] Nancy A. Lynch, "Distributed Algorithms", Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[8] K.L. McMillan, "Symbolic Model Checking: an Approach to the State Explosion Problem", Ph.D. Thesis, Carnegie Mellon University, 1992, CMU-CS-92-131.

[9] David A. Patterson, Garth A. Gibson, and Randy Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)". *Proceedings SIGMOD International Conference on Data Management,* 1988, pp. 109-116.

[10] Mandana Vaziri and Nancy Lynch. "Proving Correctness of a Controller Algorithm for the RAID Level 5 System". MIT Laboratory for Computer Science, Technical Report, December 1997. Available by anonymous ftp at `ftp://theory.lcs.mit.edu/pub/tds/raid.ps.Z`.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890